The Ferite Developers Guide 1.0 - Extending and Embedding The Ferite Engine

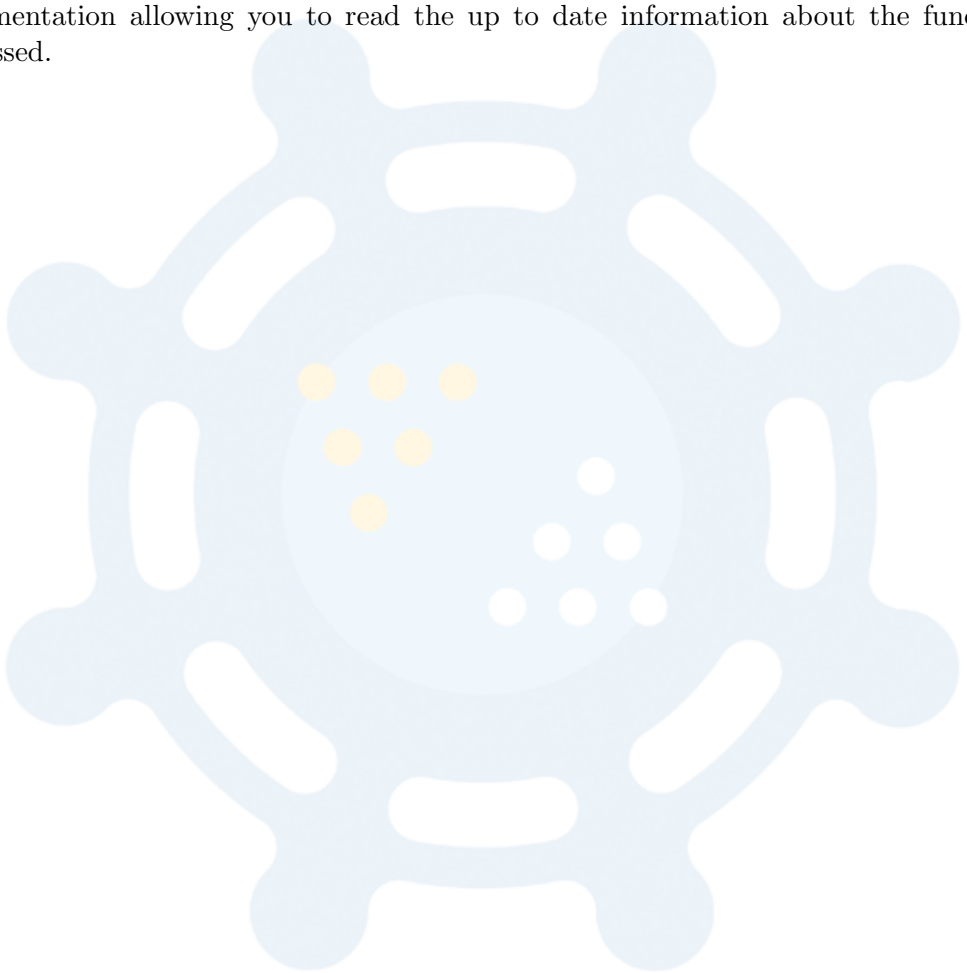
May 25, 2005

Contents

1 Introduction

It is highly recommended that you read the `ferite` manual before you continue with this manual as it relies on the fact your are fluent with the terminology and structure of a `ferite` program.

This document is provided to make it easier to do one (or all) of several things: write a ferite module, write a native module to use with ferite - both with and without using `builder` tool, accessing the internals of the ferite engine: calling functions, changing variables, creating objects etc. It is suggested that you read this with the C api documentation allowing you to read the up to date information about the functions discussed.



2 Creating Basic Modules

A module is really nothing more than a `ferite` script that resides within `ferite`s module search path. By default, the `ferite` command line tool will look for modules in the current directory and the system's `ferite` module directory (usually this is found in the directory `/usr/lib/ferite/module-source/`, although this fact can change from platform to platform). The module must have a file extension of either `.fe`, `.feh` or `.fec` in order to be recognized by ferite. The convention is that constants and the default values for a module should be stored within a `.feh` file, modules mixed with native code should have the extension `.fec`, and the normal `ferite` code in a `.fe` file. It is important to note that `ferite` treats all these files the same, it just provides the ability to have the different extensions to make the intended use of the file more obvious.

Essentially any script that you write can be included as a module. A script can import modules and other scripts by using either the `uses` keyword, or the `include()` operation. When you import a module, you refer to it by its filename, minus the .fe, .feh or .fec extension. So `mymodule.fe` would be imported by `uses "mymodule";` - `ferite` will automatically search the extensions. `ferite` will correctly resolve relative paths when importing a module.

The following example shows a script importing a module and accessing an exposed function, and the module that is imported. They are in separate files residing in the same directory.

File 1 (the importer); Name: myscript.fe

```
uses "mymodule";
foo.bar();
```

File 2 (the module); Name: mymodule.fe

```
uses "console";

namespace foo{
  function bar(){
    Console.println("Hello there!");
  }
}
```

Execution and result:

```
$ ferite myscript.fe
Hello there!
$
```

In the previous example, the module had exposed a namespace (foo), and a function within that namespace (bar). However, this is not the limit of what can be exposed.

Modules can expose functions, classes, namespaces, and global variables. Like regular scripts, modules can also modify existing namespaces and classes by using the `modifies` keyword. There is nothing special that a module must do in order to expose functionality. When a module creates a namespace, it is automatically exposed. The same goes for classes, functions and global variables.

Something that should be noted, is that any code in the anonymous function of the module will be executed when the module is first imported. You can safely put run-once initialization code in a module's anonymous function.

Here is an example of a module taking advantage of several abilities.

Name: myothermodule.fe

```
uses "console";

global {
 number gMyNumber = 7;
}

class myclass {
 string WhatISaid;

 function constructor(string WhatToSay){
  Console.println(WhatToSay);
  self.WhatISaid = WhatToSay;
 }

 function tryme(){
  Console.println("You called myclass.tryme()!");
  Console.println("When created, I said:  " + self.WhatISaid);
 }
}

namespace mynamespace {
 function hellothere(){
  Console.println("Hello there!");
 }
}

function plainfunction(){
 Console.println("You called plainfunction()!");
}

Console.println("I could be a module initializer!");
```

This code would result in `gMyNumber` being exposed as a global variable. The class `myclass` would be available, as well as all of its class members. The namespace 'mynamespace' would also become available, and it would house a single function called `hellothere`. You would also get a function called `plainfunction` placed in the main namespace, accessible simply by its name. And to top it off, upon the importing of the module, the `Console.println` statement would be executed. This is a very important feature to note, as it allows for module writers to place initialisation code that will be executed.

The next task is to cover native modules.

3 Creating Native Modules

3.1 Introduction

A native module is a `ferite` module that contains native code to interface with the surrounding system. This can be of two main forms, a mix of both native code and `ferite` script (which is how the base modules for `ferite` are written) or they can be completely made up of native code. Native code, in these examples, is C code.

Native modules can either be written by hand in the native language, written in ferite-c and converted to 100% native code, or, the prefered option, use ferite-c to build a hybrid module. This section covers the method used to create a hybrid module. These are by far the easiest to maintain and very quick and easy to build. Modules that do not require the ferite-c file at runtime are dicussed later, after `builder` tool and accessing the internals of `ferite` has been dicussed. This is due to the fact that they are harder to write and require knowledge of the `ferite` internals.

3.2 Builder

Ferite-c files (.fec) are compiled using a special tool, called `builder` which is run on the command line. `builder` is only used for the creation of native modules. It is not required in order to run pre-built native modules. Depending on your `ferite` installation, you may need to install a development package to have access to `builder`.

What does builder do?

`builder` reads a ferite-c file and creates the necessary C source, header files and automake file that will be needed to compile the module. It takes several command line parameters, only a few of which we will cover here. You can pass `builder` either `--help` or `-h` on the command line to see all of the available options. `builder`, by default, assumes that you are going to build a hybrid module and generates the code for this.

The switch we are currently most interested is -m. The -m switch allows you to specify the name of your module to `builder`. This name will be used to determine the names of the files `builder` will create while reading the ferite-c file. If you do not specify a name using -m, it will defaultto `modulename`. For simplicity we will also use the -c and -f switches, which prevent the creation of a config.m4 and Makefile.am, respectively.

Example of using `builder`:

```
$ builder -c -f -m mymodule mymodule.fec
```

When you run `builder`, it will create several output files, named according to the module name. The main files created are:

- *modulename*_core.c (holds the register, unregister, init and deinit functions)

- *modulename*_misc.c (holds native code for the anonymous/_start function, if any)
- *modulename*_header.h (holds include statements that the various c files need)
- *modulename_classname*.c (you will get one of these for every class defined in the .fec)
- *modulename_namespacename*.c (you will get one of these for every namespace in the .fec)

These files will need to be compiled into a shared object or a DLL (depending on your platform). For simplicity, we will simply refer to shared objects from here on, but they are interchangeable with DLL's. Both the resulting shared object and the original ferite-c file are needed for `ferite` to successfully import the module. You will need to place the ferite-c file in the module path, which was explained in the previous section. The shared object will need to be placed in the native search path. This is where`ferite` looks for all native modules. It is usually `/usr/lib/ferite/module-native/`*platform*, though the actual location may vary depending on the installation (ex. `/usr/lib/ferite/module-native/linux-gnu-`

Note: If you are interested in auto generation tools for standalone modules, you will probably be interested in the generate-module utility. `builder` creates input files for automake and the like specifically tailored for modules that will be included with the `ferite` source. The `generate-module` utility is geared more towards auto generation for standalone modules. The other tool that can be used is `farm`. Both these tools are covered later in this guide.

3.3 Ferite-C File Contents

Ferite-c files are very similar to basic modules. In fact you can quite easily run `builder` on a basic module. You just would end up with a lot of source files that didn't have much content. In order to get some content into those files, we need to tell `builder` what parts of our module are written in C, instead of `ferite` script. To do this, there are several new sections and keywords that we can place within our ferite-c file.

```
uses "modulename.lib"
```

One of the most important pieces of a ferite-c file, is a uses statement at the top that tells `ferite` at runtime to load the shared object file for the native module.

When you compile the files that `builder` creates into a shared object, `ferite` has no way of knowing the resulting file's name. Usually, people will compile it into a file called *modulename*.so, where *modulename* is the name of the module. However this is not required. You could quite easily compile a module from source obtained by building bob.fec, and call it jimmy.so.

The solution is to explicitly tell `ferite` to load a shared object by name. This is done with a special case of the uses statement. The syntax is much like the normal

uses statement, only you place a .lib extension on the name of the module that is to be imported. This is such that `ferite` can know to load the native library for that platform without forcing the programmer to take into account specifics of that platform.

```
uses "bob.lib";
```

This will tell `ferite` to look in the native module path for a file called bob.so on `Linux` and bob.dylib on `Mac OS X`, and to import it. This type of a uses statement can also be used within a regular `ferite` script to load a native only module.

3.3.1 module-header

The module-header section is where you will place any #include statements, or #define statements, or anything else that you expect your native code will need. The syntax for creating a module-header in a ferite-c file is much like defining a global section in a regular script. The code that is declared within the `module-header` is availible in all generated C files.

For example:

```
module-header {
  ...your headers go here...
}
```

Anything you place within the module-header section will be placed in the *module-name*_header.h file when `builder` parses the ferite-c file. This header is then included in every C source file that `builder` creates. You can have as many `module-header` blocks, the code will just be all placed together in the header file.

Here is an example of a module-header:

```
module-header {
  #include <stdio.h>
  #include "utility.h"
}
```

`builder` doesn't do any validity checking in between the curly braces. So if you have typographical errors, you probably won't know until you try to compile the module.

3.3.2 module-init

This section allows you to specify native code that is executed when the module is loaded into a script. It is an optional section, but `builder` will create an empty module-init function in the C source. This function will be executed when the uses "modulename.lib" is executed.

The module's _start function (sometimes referred to as the anonymous function) is also executed when it is first imported, but module-init code is executed first. Also, the _start function cannot contain native code. So if your module initialization requires multiple jumps between native and `ferite` code, you can use the _start function to call native functions where necessary and use `ferite` code for everything else.

The syntax for creating a module-init section is similar to module-header:

```
module-init {
  ...your code goes here...
}
```

This will cause all of the code placed within the curly braces to be placed in the module's init function. In case you're interested, the build destination is the *modulename*_core.c file, in a function called *modulename*_init(). The function returns void and has 1 parameter, "FeriteScript *script", which is accessible to the code within the section.

3.3.3 module-deinit

This section is syntactically almost identical to the module-init section. Like module-init, module-deinit is not a required section. Again, `builder` will create empty module-deinit function in the C source for you if you do not specify one. This function is called when a script is being deleted.

Code in this section is executed when the script that loaded the module is being deleted. More precisely, it is run by a call to ferite_script_delete(). However, you usually don't have to worry about the specifics unless you're embedding `ferite` in your application. For most purposes, just know that this code is run when the script has finished executing.

Here is an example of a module-deinit section:

```
module-deinit {
  ...your code goes here...
}
```

As you can see, it is basically the same as module-init. The return type is void, so you shouldn't try returning anything from this function. It also has the affected script passed into it, which is accessed exactly the same as you would for module-init.

3.3.4 module-register and module-unregister

When a native module's shared object is loaded, its register function is called once. This allows the shared object to setup any system specific things. Symetrically, module-unregister is only called once, and that is when the `ferite` module system decides to

unload the shared object. They are both blocks of code like module-init and module-deinit and should be used the same way.

3.3.5 Native Functions, the `builder` way

When developing a native module with `builder` it will be necessary to create functions that can be called by `ferite` scripts. To make this easy there are only two main differences between a `ferite` function and a native function. These are the keyword `native` and that the bodies of the functions are written in C.

First we'll start with an example of how to declare a simple native function:

```
native function foo() {
  ...your code goes here...
}
```

This would result in the C source between the curly braces being placed in one of the C source files. The exact file and the exact function name created depends on the namespace or class that the function is declared in. This might vary from version to version so I won't get into it here, but feel free to look at the source created. You'll probably be able to figure it out from there. To a scripter the function looks and tastes the same as a normal `ferite` function.

It should be noted that within each function the following variables are accessable:

- `script` - a pointer to the FeriteScript in which the function was called.
- `function` - a pointer to the FeriteFunction which owns the function executing.
- `params` - the null terminated list of parameters (see Calling Functions for more information).
- `self` - **Note!** for a function that is an object function, selfwill point to the FeriteObject*, for a namespace it will point to the FeriteNamespace* that the function lives in, for a class function it will point to the FeriteClass* the function lives in.
- `current_yield_block` - A pointer to the current closure that may have been passed into the function. The `deliver` keyword, that is used within a ferite script, uses this closure and calls the `invoke` function on it.

Parameters

The next step is to pass in some variables, and it is pretty easy to do. Simply declare the variables as you would normally do for any `ferite` script. When you get inside of the function, the values passed in will be converted to units that are workable in C with the same name. Complex objects will be presented to you in the form of pointers to different types of structs according to their type. All variables are available by the names you

gave in the function declaration. Following is a quick breakdown of the different types and how they convert.

- number - Numbers are converted to doubles because doubles can represent LONG_MAX, and `ferite` numbers support floating point values anyways. If you expected to use the value as an integer in your function you can simply cast the double to a long. It is a good idea to check that the number passed in is not greater then LONG_MAX before you cast it to an long, otherwise you might end up with some funny looking results.
- string - Strings are converted to FeriteString *, and their C-string values are accessible by struct element 'data'. So you can retrieve the value of string mystring by mystring->data. Following is an example that accesses a string's value by using it in a call to `strdup`.

```
native function foo( string mystring ){
  char *mystring_copy = strdup( mystring->data );
  int length = mystring->length;
}
```

- object - Objects are instances of classes and represented within C code as FeriteObject's, which must be accessed by reaching into `ferite`'s internals. This is covered in the next section "Accessing Ferite Internals".
- array - Arrays are represented by a FeriteUnifiedArray* and must also be accessed by reaching into `ferite`'s internals.
- void - A void variable has no conversion and the name is just a pointer to a variable.

The example below illustrates what builder generates given a the function given.

```
native function example( number x, string s, object o, array a, void
v )
{
  printf( "Value of x:  %f\n", x );
  printf( "String contents:  %s\n", s->data );
  printf( "Object reference count:  %d\n", o->refcount );
  printf( "Array size:  %d\n", a->size );
  printf( "Type:  %d\n", v->type );
}
```

This is the C code that is generated:

```
FE_NATIVE_FUNCTION( ferite_module_example_nsoav )
{
 double x;
 FeriteString *s;
 FeriteObject *o;
```

```
 FeriteUnifiedArray *a;
 FeriteVariable *v = params[4];
 FeriteNamespace *self = FE_CONTAINER_TO_NS;

 ferite_get_parameters( params, 5, &x, &s, &o, &a, NULL );

 { /* Main function body.  */
#line 5 "test.fec"

   printf( "Value of x:  %f\n", x );
   printf( "String contents:  %s\n", s->data );
   printf( "Object reference count:  %d\n", o->refcount );
   printf( "Array size:  %d\n", a->size );
   printf( "Type:  %d\n", v->type );

 }
 FE_RETURN_VOID;
 self = NULL;
}
```

The important point being made with this example is how the parameters are automatically converted from FeriteVariable*s to their real types. This makes writing native functions much easier.

Return Values

Now that we have discussed handling parameters, next you need to know how to return values from functions. Any time you don't specify a return value and your function runs off the end of its scope, `builder` will assume you meant to return void and will automatically insert a call to return nothing. This can been seen in the previous example, where, at the end of the generated code, the following can be seen:

```
FE_RETURN_VOID;
```

If returning void is not the desired effect, or you would like to specify a position to return from other than running off the end of the function's scope, you will need to specifically return a value using one of the following C macros. The macros are designed to take a value from the C code, wrap it up into a FeriteVariable, do a little house keeping and then return from the function.

- `FE_RETURN_VOID` - returns void to the caller, this is synonymous with not returning anything.
- `FE_RETURN_TRUE` - returns true to the caller.
- `FE_RETURN_FALSE` - returns false to the caller.

- `FE_RETURN_LONG( value )` - returns a number to the caller with the contents of the given long.
- `FE_RETURN_DOUBLE( value )` - returns a number to the caller with the contents of the given double.
- `FE_RETURN_STR( string, freeme )` - returns a FeriteString* to the caller. The parameter "string" is passed in as a FeriteString*. If freeme == FE_TRUE, string is freed using `ferite`'s memory manager. if freeme == FE_FALSE, it is not freed at all.
- `FE_RETURN_CSTR( string, freeme )` - returns a char* to the caller. The parameter "string" is passed in as a char*. If freeme == FE_TRUE, string is freed using `ferite`'s memory manager. if freeme == FE_FALSE, it is not freed at all.
- `FE_RETURN_ARRAY( pointer to array )` - returns an array to the caller.
- `FE_RETURN_OBJ( pointer to object )` - returns an object to the caller (objects are instances of classes).
- `FE_RETURN_NULL_OBJECT` - returns a null object to the caller (useful for functions that are expected to return an object, but need to signify an error condition).
- `FE_RETURN_VAR( variable )` - returns a FeriteVariable to the caller. This allows you to return a variable that you have created yourself to the engine. It will tag the variable allowing `ferite` to clear it up when it is finished with. If you want to return a variable, but keep hold of it, you must simply return the variable as you would an item from a normal c function. eg:

```
return someVar;
```

All of these macros actually convert the given return values into a FeriteVariable * which is then returned to the caller. As a general rule, you should always use the available macros when mixing C and `ferite` to prevent your functions from breaking if the interface ever changes. These macros will be kept up to date, so you are safe to use them. All the macros tag the variable they create as being disposable; this is a delayed deletion mechanism that gives `ferite` permission to free up the variables it has finished using. This is important to note, especially with `FE_RETURN_VAR` because you may not want the variable cleaned up.

And Finally

The previous few sections should give you enough information to get you up on your feet and writing native functions. To play with `ferite`'s internals you will need to read on where this is dicussed in depth. However, here is an example:

```
uses "Example.lib";

namespace Example {
```

```
  native function Add( number left, number right ) {
    FE_RETURN_DOUBLE( left + right );
  }
  native function Error( number code, string error ) {
    fprintf( stderr, "Error:  %d:  %s\n", (int)code, error->string
);
    FE_RETURN_LONG(-1);
  }
}
```

3.3.6 Classes and Namespaces

Classes and namespaces in native modules are created exactly like their non-native counterparts. You simply declare the namespace or class in the ferite-c file, and when a script tells `ferite` to import the module, `ferite` will parse the .fec and create the namespaces and classes as usual and link up the native functions from the shared object. There is absolutely no syntax change for creating classes and namespaces. Pretty straight forward isnt it?

There is, however, the added ability to place native functions within classes and namespaces. The syntax for doing so is no different that what you've already seen, just place them within the curly braces of the namespace or class that you would like them to be a part of.

Here is an example of a native function in a namespace:

```
namespace foo {
  native function bar() {
    ...your code goes here...
  }
}
```

And here is an example of a native function in a class:

```
class foo {
  native function bar() {
    ...your code goes here...
  }
}
```

You can also make functions in classes static, as was described in the user manual. So of course we can make those native functions as well. Simply place the keyword static in the function declaration.

```
class foo{
  static native function bar1(){
```

```
    ...your code goes here...
  }
  native static function bar2(){
    ...your code goes here...
  }
}
```

Both bar1() and bar2() are native functions that are static within the class foo. The order of the keywords does not matter.

Self Data

When a native function belonging to an object, or namespace or class, is called, there is the `self` variable available. This is a pointer to the FeriteObject/FeriteClass/FeriteNamespace which is currently executing. Now, to make life easier there is a part of the structure that allows you, the module writer, to attach any data to it. This is called (and refered to) as `odata` and is short for object data. `ferite` does not and will never touch this. It is the job of the programmer to deal with it. It is very simple to use, simply access the odata member on self.

```
self->odata = get_some_resource();
```

Example: In the 'filesystem' module, odata is used to store a pointer to the FILE* pointer for file streams.

```
#define SelfObj (FILE*)(self->odata)
```

3.3.7 Finally

This section should have helped you get off your feet and understand the way in which `builder` can help you not only rapidly develop modules but keep them very close to `ferite` code. You should look at the `.fec` files that ship with `ferite` to clarify any doubts you have.

3.4 Without Builder

The natural flow of this document means that creating native modules without `builder` should be dicussed here. As most of the dicussion requires knowledge dicussed in the next section, this will be left until after the internals of `ferite` have been looked at.

4 Accessing Ferite Internals

4.1 Introduction

This section is designed to teach you how to access, modify, create, and destroy various structures within `ferite`. It covers variables, functions, classes, and namespaces. It will first cover very basic memory management, then cover variables, namespaces, calling functions, calling object and class functions, and creating a class.

It should be noted that this section will cover the registering and accessing of methods, but won't tell you how to write one from scratch manually. That will be left for the next section where native modules by hand will be dicussed.

4.2 The Memory Manager

Under normal operation `ferite` uses its own memory manager, which is basically a sub allocator, to achieve some significant performance gains over the standard malloc/free operations. This memory manager is used throughout `ferite`, and the data that is passed around in `ferite` is expected to be allocated under this manager. The rule of thumb is that any memory that touches ferite should be obtained from ferite or placed in the odata part, otherwise, it is more than likely that `ferite` will try and free some memory and crash.

This memory manager acts much like malloc/free in terms of how you use it. There are functions that mirror the malloc, calloc, realloc, and free calls.

- fmalloc(size)
- fcalloc(size, blocksize)
- frealloc(ptr, size)
- ffree(ptr)

These functions all act like the functions they replace. However, they are different so don't mix calls on memory between malloc/free and fmalloc/ffree. They play well in the same sand-box, but don't ask them to swap Tonka trucks with each other.

4.3 Working With Variables

4.3.1 Accessing a Variable's Data

`ferite` internally represents all variables using FeriteVariable *'s, and they represent any native type within `ferite`. `builder`, and the return macros you've already seen

are in place to perform conversions for the sake of convenience. But sometimes you just need to stick your fingers in the pudding and get dirty.

There are a few bits of general information you can get from a FeriteVariable * without looking specifically into on variable type. The internal variable name is accessible (it is a null terminated C string) and usually the variable name has been automatically generated by an operator or a function. It's main use within the engine is to store the hash key the variable has within an array. But if you'd really like to have it, you can access is by:

```
var->name
```

Much more useful than the variable name is the variable type. This will tell you if the data held is a number (and what kind), a `ferite` string, an object, an array, a class, namespace, or void (nothing at all). It is accessible by:

```
var->type
```

It is an integer that can be any one of the following values:

- `F_VAR_VOID` - a void variable, no value.
- `F_VAR_LONG` - a number variable as a C long.
- `F_VAR_DOUBLE` - a number variable as a C double.
- `F_VAR_STR` - a string variable.
- `F_VAR_UARRAY` - an unified array variable.
- `F_VAR_OBJ` - an object.
- `F_VAR_CLASS` - a class.
- `F_VAR_NS` - a namespace.

There are a number of additional macros available for accessing the actual data within different variable types. You should use these macros as much as possible when working with `ferite` variables. Internal structures may change, but these macros should always be up to date and provide exactly the same semantics when it comes to value access.

- `F_VAR_VOID` - since this is a void variable, there really isn't any data to gain access to.
- `F_VAR_LONG` - the data can be accesses by VAI(var) . This will make it act exactly like a C long. You can read its value and set new values.

  Example:

  ```
  VAI(mynum) = 7;
  ```

- `F_VAR_DOUBLE` - the data can be accessed by VAF(var). This will make it act exactly like a c double. You can read its value and set new values.

  Example:

  ```
  VAF(mynum) = 8.16;
  ```

- `F_VAR_STR` - using the VAS(var) macro will get you a FeriteString*, which can then be used in API functions to perform various operations. You can access the string's length and data by using the original FeriteVariable * in the FE_STRLEN(var) and FE_STR2PTR(var) macros, respectively. FE_STR2PTR behaves like a char*, and FE_STRLEN behaves like an int. Whenever you change a string's content, you must always update its internal size to reflect the new actual size.

  Example:

  ```
  ffree(FE_STR2PTR(var));
  FE_STR2PTR(var) = fstrdup("My new string!");
  FE_STRLEN(var) = strlen(FE_STR2PTR(var));
  ```

  There are a whole host of functions within the `ferite` engine for manipulating FeriteString*'s allowing you to do comparisons and replacements on the strings. It should also be noted that FeriteString*'s are designed to hold binary data.

- `F_VAR_OBJECT` - using the macro VAO(var) will get you a FeriteObject*, which can then be used in a variety of API functions to access variables and functions within that object.

- `F_VAR_UARRAY` - using the macro VAUA(var) will get you a FeriteUnifiedArray *, which can then be used in the unified array API functions to add, retrieve and remove values from the array.

- `F_VAR_CLASS` - using the macro VAC(var) will get you a FeriteClass*, which can then be used in a variety of API functions to access variables and functions within that class. To get a class passed into a function you must, currently, use the `void` type.

- `F_VAR_NS` - using the macro VAN(var) will get you a FeriteNamespace*, which can then be used in a variety of API functions to access variables and functions within that namespace. To get a class passed into a function you must, currently, use the `void` type.

4.3.2 Changing a Variable's Type

To change a variable from one type to another you need to call the function `ferite_variable_convert_to_ty` This takes the script, the variable to change and the new type that you require. The function will take care of any arrays, objects etc that may still be attached to the variable. For example, to change a double to a long, you can simply do this:

```
ferite_variable_convert_to_type( script, var, F_VAR_LONG );
VAI(var) = 10;
```

And to change it back:

```
ferite_variable_convert_to_type( script, var, F_VAR_DOUBLE );
VAF(var) = 10.0;
```

It is considered bad for you to simply change the type of a variable. It is not encouraged at all. It is therefore on your own head to keep things correct.

4.3.3 Creating and Destroying Variables

Creating variables is quite simple: each variable type has a 'create' function that returns a FeriteVariable*. You already know how to manipulate these variables (or at least get to the information needed to manipulate them), so I'll just quickly run through the variable types available, and their creation functions. The parameters should be pretty self-explanitory, if not please refer the the C API document. It should be noted though that they all take the same argument "alloc." This tells `ferite` whether or not the name of the variable should be allocated or whether it is static; You should state that the name is static if you are passing a string constant so that ferite wont waste memory. E.g.

```
char *name = strdup("SomeName");
FeriteVariable *copy_name =
  ferite_create_number_long_variable( name, 42, FE_ALLOC );
FeriteVariable *ref_name =
  ferite_create_number_long_variable( "SomeName", 42, FE_STATIC );
free(name);
```

In the above example, the first allocation of the variable causes `ferite` to copy the name, allowing the `name` variable to be free'd up. The second variable allocation tells ferite to retain a reference to the name because we know it wont be cleared up.

- `F_VAR_VOID` - FeriteVariable *ferite_create_void_variable(char *name, int alloc);
- `F_VAR_LONG` - FeriteVariable *ferite_create_number_long_variable(char *name, long data, int alloc);
- `F_VAR_DOUBLE` - FeriteVariable *ferite_create_number_double_variable(char *name, double data, int alloc);
- `F_VAR_STR` - FeriteVariable *ferite_create_string_variable(char *name, FeriteString *data, int alloc);
- `F_VAR_STR` - FeriteVariable *ferite_create_string_variable_from_ptr(char *name, char *data, int length, int encoding, int alloc); Currently, the encoding value is always

FE_CHARSET_DEFAULT. The reason for it being set now is so the in the future when the encoding of a string is important code will still work unmodified.

- F_VAR_UARRAY - FeriteVariable *ferite_create_uarray_variable(char *name, int size, int alloc);
- F_VAR_OBJ - FeriteVariable *ferite_create_object_variable(char *name, int alloc);
- F_VAR_CLASS - FeriteVariable *ferite_create_class_variable(FeriteScript *script, char *name, FeriteClass *klass, int alloc)
- F_VAR_NS - FeriteVariable *ferite_create_namespace_variable(FeriteScript *script, char *name, FeriteNamespace *ns, int alloc)

To delete any ferite variable, you use the ferite_variable_destroy function. This function takes the current script and a FeriteVariable * as parameters, and it returns void.

```
void ferite_variable_destroy(
 FeriteScript *script,
 FeriteVariable *var );
```

You can use this function on any type of variable. Each will be handled in the appropriate manner according to its type. Strings will have their C string data freed by ffree and will then be destroyed. Objects will have their destructor called before they are destroyed. Lastly, unified arrays will have the variables at each of its indexes destroyed in the appropriate manner according to their type and will then, themselves, be destroyed.

4.4 Working With Namespaces

In this section we'll cover how to create and delete namespaces, and how to create, access, and delete variables and how to register and delete functions, and how to find things within them. Namespaces are created by registering them within the script. This can be done with the following function:

```
FeriteNamespace *ferite_register_namespace(
 FeriteScript *script,
 char *name,
 FeriteNamespace *parent )
```

The function takes three parameters: the script to register the namespace into, the name of the namespace you wish to create, and the parent where you wish to create the new namespace. The parent must be a valid pointer to a FeriteNamespace, you can either find one with ferite_find_namespace, or you can simply use script->mainns to use the top-level namespace of a script as the parent. If the register is successful, the

FeriteNamespace * that refers to the new namespace is returned. The data it points to is internally allocated, so do not destroy it. If the register failed, it will return NULL.

```
module-init {
  FeriteNamespace *mobile =
    ferite_register_namespace( script,
         "Mobile", script->mainns );
}
```

Once you have a namespace created, you can delete it with this function:

```
int ferite_delete_namespace( FeriteScript *script, FeriteNamespace
*ns )
```

This will destroy the namespace after recursivly destroying all of its children. This includes all variables, sub-namespaces, classes and functions. It currently always returns 1.

Creating and deleting namespaces is only fun for a short while. Eventually you'll want to put variables into your new namespace, and probably functions and classes as well. The next three functions will allow you to do just that.

```
FeriteVariable *ferite_register_ns_variable(
        FeriteScript *script,
        FeriteNamespace *ns,
        FeriteVariable *var )
```

This will register a variable into the namespace that you provide. If you've recently created the namespace, you can use the FeriteNamespace * that the register function returned. Otherwise you will have to look up the FeriteNamespace * to the namespace you wish to place your variable in using the ferite_find_namespace function. The value returned is always the same as the value passed in as the var parameter. The variable will be accessible under the new namespace according to its name stored in the FeriteVariable struct. So you might want to make sure you set it to something intelligent before you register it into a namespace. E.g.

```
module-init {
  FeriteNamespace *mobile =
    ferite_register_namespace( script,
              "Mobile",
              script->mainns );
  FeriteVariable *signal = ferite_create_number_long_variable(
              "signal",
              0,
              FE_STATIC );
  ferite_register_ns_variable( script, mobile, signal );
}
```

```
FeriteFunction *ferite_register_ns_function(
        FeriteScript *script,
        FeriteNamespace *ns,
        FeriteFunction *f )
```

This functions registers a function into the given namespace. The return value is always the same as the value passed in as the f parameter. Again, the name of the element comes from the name field of the FeriteFunction struct. Set it before you register the function.

```
FeriteClass *ferite_register_ns_class(
        FeriteScript *script,
        FeriteNamespace *ns,
        FeriteClass *klass )
```

This will register a class into the given namespace. The return value is always the same as the value passed in as the klass parameter. Once again, the name of the element comes from the name field of the FeriteClass struct. Set the name before you register the class. Most of the time you will never use this as the standard way to create a class will also automatically register it, it is merely mentioned here for completness.

The next logical step is gaining access to variable, functions, and classes that are registered to namespaces. This is done by retrieving a `FeriteNamespaceBucket` which contains the information you desire in its data element. The following function is used for retrieving these buckets:

```
FeriteNamespaceBucket *ferite_find_namespace(
        FeriteScript *script,
        FeriteNamespace *parent,
        char *obj,
        int type )
```

This will return a FeriteNamespaceBucket * on success, or NULL on failure. It takes a script, and a starting point as the first two parameters. The third parameter is the dot-delimited name of the object you are looking for, relative to the parent namespace given. So if you are using the root namespace (script->mainns) as your parent namespace, and wish to access mynamespace.myothernamespace.myvar, then you would pass "mynamespace.myothernamespace.myvar" as the third parameter. However, if you already have a FeriteNamespace * that refers to 'mynamespace', then you could pass that in as the parent (2nd parameter) and then access myvar by passing "myothernamespace.myvar" as the obj (3rd parameter). Lastly, if you already have the FeriteNamespace * for 'myothernamespace', then you would simply pass "myvar" as the obj. Because you are only dealing with one level of depth, you do not place a period within the obj in that instance. The fourth, and last, parameter is the type of object you are looking for. It is always one of the following defined types:

- `FENS_NS` - retrieves namespaces

- FENS_VAR - retrieves variables
- FENS_FNC - retrieves functions
- FENS_CLS - retrieves classes

If you choose to pass 0 to the function, you will get back the named FeriteNamespaceBucket if it exists. Using the above defines allows you to tell ferite_find_namespace what type of bucket you are looking for guaranteeing that what you get back is the correct item and type.

Again, once you have the bucket, you can access the desired value by looking in the data element. Example:

```
FeriteVariable *myvar = NULL;
FeriteNamespaceBucket *nsb = NULL;

nsb = ferite_find_namespace(script,
  script->mainns,
  "mynamespace.myvar", FENS_VAR);

if( NULL != nsb ){ /* we found it!  */
  myvar = (FeriteVariable *) nsb->data;
  /* we needed to cast because nsb->data is a void* type */
}
```

At this point I can use myvar just like any other FeriteVarible *, because it is one! When the value of this variable is changed it will be noticable straight away within the script. It is also important to note that you must not take these variables you have obtained and return them to the script via FE_RETURN_VAR. This will cause `ferite` to delete the variable and leave dangling pointers. If you wish to return the variable simply return it like you would a normal c variable:

```
return myvar;
```

To get a function is the same process. Example:

```
FeriteFunction *func = NULL;
FeriteNamespaceBucket *nsb = NULL;

nsb = ferite_find_namespace( script,
  script->mainns,
  "mynamespace.function",
  FENS_FNC );

if( NULL != nsb ){
  func = (FeriteFunction*)nsb->data;
```

```
  ....
}
```

It is good to note that within `ferite`'s source, the convention is to call the namespace bucket variable 'nsb'.

As promised at the beginning of this section, here is how to unregister elements from namespaces:

```
void ferite_delete_namespace_element_from_namespace(
        FeriteScript *script,
        FeriteNamespace *ns,
        char *name )
```

This will delete the element name from the namespace ns within the script script. Be careful though, this function will not burrow down layers of namespaces to find the element you specify. So you cannot use the dot notation here, this is a deliberate design choice to stop accidental deletion of the wrong elements. You must first find the immediate parent of the element (using ferite_find_namespace), and pass that in as the namespace ns. You can use this to delete namespaces from within namespaces as well, and in that case it will also recursively destroy the deleted namespace's contents.

So that is all there really is to namespaces. They are an excellent form of container both in and out of scripts!

4.5 Working With Objects And Classes

4.5.1 Creating Classes

Registering classes is much the same as registering namespaces. You first register the class, then you add the variables and functions you wish to publish in them.

To register a class you use the ferite_register_inherited_class function call. This will create the class, setup the inheritence, register the class wihin a namespace for you and return it in one fell swoop.

```
FeriteClass *ferite_register_inherited_class(
        FeriteScript *script,
        FeriteNamespace *ns,
        char *name,
        char *parent )
```

The first parameter is the script, the second is the namespace in which you want to place the class, the third is the name of the class by which programmers can reference it and the fourth is the name of the class the new class inheirts from. The fourth argument can be in standard dot notation and is the name of the parent class. For instance, it could be "Sys.Stream". The function will start looking for the class in the namespace that is

passed to the function, and then start in the top level script namespace. For instance, if the "Sys" namespace was passed to the function, you would want to specify "Stream". If you do not wish for your class to inherit from any existing class simply pass NULL and the new class will be automatically placed as a subclass of the base class "Obj".

Registering variables and functions with a class is much the same as registering them with a namespace, you simply pass an extra parameter to say whether or not the item is static (linked to the class) or an instance variable (linked to the object created from the class).

To add a variable you call:

```
int ferite_register_class_variable(
        FeriteScript *script,
        FeriteClass *klass,
        FeriteVariable *variable,
        int is_static )
```

The second argument is the class to add the variable to. The class can be obtained from creating a new class or pulling one out of a namespace. The third argument is the variable to add. The fourth variable is whether or not the variable is static.

To add a function you call:

```
int ferite_register_class_function(
        FeriteScript *script,
        FeriteClass *klass,
        FeriteFunction *f,
        int is_static )
```

The arguments are almost identical except for the third one which is a pointer to a `ferite` function.

4.5.2 Creating Objects

Creating objects is very straight forward. There are two main method calls that can be used.

The first is `ferite_build_object`. Its pupose is to simply allocate a FeriteVariable*, allocate the necessary structures (such as it's instance variables, and pointers to functions) and add it to the `ferite` garbage collector. `ferite_build_object` does **not** call the new object's constructor. This is very useful for when you are doing manual setup of an object. The prototype for the function is:

```
FeriteVariable *ferite_build_object(
        FeriteScript *script,
        FeriteClass *nclass )
```

The second is `ferite_new_object` which does all the same things `ferite_build_object` does except it will call the constructor for the new object. It will return an FeriteVariable* that is ready to be cleaned up by `ferite` as and when it is returned to the engine and no longer wanted. It has the prototype:

```
FeriteVariable *ferite_new_object(
        FeriteScript *script,
        FeriteClass *nclass,
        FeriteVariable **plist )
```

The first two arguments are the same for `ferite_build_object`, the current script and the class you wish to instantiate. The third argument is the parameter list to be passed to the object's constructor. Read the next section on calling functions to find out how to create one and what they consist of.

4.5.3 Accessing Variables

Firstly, we'll cover how to access variables within objects and classes. It is done essentially the same way for each. Both FeriteClass and FeriteObject structs have a variables element that is a hash of all variables within them. To make life slightly easier and code more understandable there are a couple of functions for retrieving the variables from either a class or an object.

```
FeriteVariable *ferite_object_get_var(
        FeriteScript *script,
        FeriteObject *object,
        char *name )
```

This is for getting the value out of an object. It should be noted that the second argument is not a FeriteVariable* but a FeriteObject*. This means that is it necessary, if you have a FeriteVariable* pointing to an object, to call it with VAO(nameOfVariable).

```
FeriteVariable *ferite_class_get_var(
        FeriteScript *script,
        FeriteClass *klass,
        char *name )
```

Both the above functions take the name of the variable to obtain and will return a pointer to the variable if it exists, or will return NULL if it doesn't.

For example, for objects you would do this: (assume that some_object is of type FeriteVariable *, and it is a valid object)

```
FeriteVariable *myvar = ferite_object_get_var(script,
                    VAO(some_object),
                    "myvar");
```

If myvar is not NULL, then it was successfully retrieved. If you want to do the same with a class, you do this: (assume that some_class is of type FeriteClass *, and it is a valid class)

```
FeriteVariable *myvar = ferite_class_get_var(script,
                some_class,
                "myvar");
```

Again, if myvar is not NULL, it was successfully retrieved.

4.5.4 Accessing Functions

Getting functions from objects or classes is easy if you can get a variable from them (Hint: make sure you read the last section!).

To get your hands on a function in an object you simply use the function call `ferite_object_get_function`. Suprised? You shouldn't be. It looks, feels and tastes very similar to `ferite_object_get_var` except this time you get a function not a variable.

```
FeriteFunction *ferite_object_get_function(
        FeriteScript *script,
        FeriteObject *object,
        char *name );
```

To get your hands on a function tucked away in a class you simply need to use the function call `ferite_class_get_function`.

```
FeriteFunction *ferite_class_get_function(
        FeriteScript *script,
        FeriteClass *cls,
        char *name )
```

4.6 Calling Functions

Once you have a FeriteFunction *, the next thing you're probably going to want to do is call the function it to which it refers. This is one of the trickier things to do in `ferite`, but only because it involves several stages in order to complete.

Firstly, you need your FeriteFunction *, which can be obtained by using the ferite_find_namespace function. Then you'll need to create a parameter list that you wish to pass to the function. This is done with the following function:

```
FeriteVariable **ferite_create_parameter_list_from_data(
        FeriteScript *script,
        char *format,
        ...  );
```

This function does its best to make creating parameter lists simple. The first parameter is the script, the second is a format string that describes the types of variables that will make up the argument list, and the rest of the parameters are the values to be used as described by the format string. The format string must be zero or more of the following:

- n - a number, the value passed must be a C variable of type double
- l - a number, the value passed must be a C variable of type long
- s - a string, the value passed must be a pointer to FeriteString
- o - an object, the value passed must be a pointer to a FeriteObject
- a - an array, the value passed must be a pointer to a FeriteUnifiedArray

The function will return a parameter list (FeriteVariable **) which can then be used as a parameter in the next function to be dicussed. For your information, a parameter list is simply a NULL terminated C array of FeriteVariable* - these are easy to create by hand, but this function simply aids the creation.

```
FeriteVariable *ferite_call_function(
        FeriteScript *script,
        void *container,
        FeriteObject *block,
        FeriteFunction *orig_function,
        FeriteVariable **params )
```

This function will call the function and return a FeriteVariable *, which will be the returned value of the called function. It must be caught and destroyed, or you will leak memory. Even functions returning void will return a fully allocated FeriteVariable * of type F_VAR_VOID.

The first parameter is the script, the second is the pointer to the container of the function, for instance for a namespace function a FeriteNamespace, for a class a FeriteClass pointer and an object, a FeriteObject pointer. The third argument is the closure that can be passed to the function, this can be NULL. The fourth argument is the FeriteFunction you wish to call, and the last is the parameter list you had created with the previously described ferite_create_parameter_list_from_data() function. The parameter list may be NULL indicating that the function takes no arguments.

When you are finished with the parameter list, simply delete it with this function:

```
void ferite_delete_parameter_list(
        FeriteScript *script,
        FeriteVariable **list );
```

So there you have it, three steps to calling another function within `ferite`. Here is a complete example which calls 'Console.println' with the string 'Hello World':

```
FeriteFunction *println = NULL;
FeriteVariable **params = NULL;
FeriteVariable *rval = NULL;

/* Create a string to pass to the function */
FeriteString *hello = ferite_str_new( "Hello World", 0, FE_CHARSET_DEFAULT
);

/* Find the function in the scripts main namespace */
FeriteNamespaceBucket *nsb = ferite_find_namespace(
             script,
             script->mainns,
             "Console.println",
             FENS_FNC );
FeriteNamespaceBucket *console = ferite_find_namespace(
             script,
             script->mainns,
             "Console",
             FENS_NS );

if( NULL != nsb ) /* Check to see if we have the function ...  */
{
 println = nsb->data;

 /* Create the parameter list */
 params = ferite_create_parameter_list_from_data( script, "s", hello
)

 /* Call the function */
 rval = ferite_call_function( script, console, NULL, println, params
);

 /* And finally clear up after ourselves */
 ferite_delete_parameter_list( script, params );
 ferite_variable_destroy( script, rval );
 ferite_str_destroy( script, hello );
}
else
 /* We don't..  let's print an error!  */
 printf( "Cant find 'Console.println'!  Is the console module loaded?\n"
);
```

The only difference with calling a class or object function compared to that of a namespace function is the passing in of the container. There are no other special tricks that

are required.

4.7 Raising Exceptions and Reporting Errors

There are times when things go wrong. It's a painful time, but it need not be. Ferite provides a means of raising exceptions to force a programmer to deal with errors but also a means of quietly setting the error information allowing the programmer to check for non-fatal things.

It is considered good form to return error values from a function call. This is the route you should take if you require the reporting of errors. For instance, if you have a function that connects to a resource and returns an object to interact with that resource, it makes sense to return a null object (`FE_RETURN_NULL_OBJECT`) if that resource can't be obtained.

Sometimes it is not possible to return an error value. In these situations it is considered good form to use the function `ferite_set_error` [it's prototype is below]. This sets the `err` script object's values, but does not raise an exception. This allows the programmer to ignore things if needs be. It takes a number of parameters, the first is the script you are running in, the second is the error number and the last is the format of a string [same as `printf`] describing the error that has occured. It should be documented that this is the case such that the programmer knows what to look for.

```
void ferite_set_error( FeriteScript *script, int num, char *fmt,
...  );
```

When all hope is lost, there are times when an exception needs to be rasied because some has gone completely wrong. This is done by calling `ferite_error`. You can pass it the error number and the message just like `ferite_set_error`.

```
void ferite_error( FeriteScript *script, int num, char *fmt, ...
);
```

Sometimes it is nice to warn people about not so bad things, and as such there is a function `ferite_warning` which will place a warning on the script.

```
void ferite_warning( FeriteScript *script, char *errormsg, ...  );
```

4.8 Executing Code Snippets

Sometimes it is easier to execute a block of code, from within a function, written in `ferite`. For this you can use the eval mechanism. What this does is the same as the `eval` operator in `ferite`. It will compile and execute the script and then return the return value of the main function. For example:

```
rval = ferite_script_eval( script, "Console.println('Hello World');"
);
```

You must destroy the return value using `ferite_variable_destroy` just as you would a function call.

5 Native Modules - By Hand

The aim of this section is to show you how to write modules by hand. This section is also very useful for people wanting to embed `ferite` as it shows how to export an API by hand.

5.1 Functions

`builder` makes the following completely automatic. As with normal C functions, we have to declare our native `ferite` functions. This is done in three stages. First we declare the function, then we create our FeriteFunction structure and then we register it with the `ferite` engine. To declare the variable, you use the macro `FE_NATIVE_FUNCTION`, this is true for both object/class methods and normal namespace functions. This takes one argument, which is the name of the function you wish to create. After the macro, you simply write the body of your function as you normally would. For example:

```
FE_NATIVE_FUNCTION( printfnc )
{
  printf( "We are in our native function!\n" );
}
```

The next thing we need to do is create a FeriteFunction structure with which we can register the function (using the functions mentioned in the last section). This is a call to`ferite_create_external_function`.

```
FeriteFunction *ferite_create_external_function(
        FeriteScript *script,
        char *name,
        void *(*funcPtr)(FeriteScript*,FeriteFunction*,FeriteVariable**),
        char *description );
```

This takes the current script, the name of the function, a pointer to the function, and its signature description. The third means you simply pass the name of the native function, eg. in the above example it would be `printfnc`. The description is slightly more complicated. It is a null terminated string which takes a number of characters that describe what arguments the function can take.

- n - number
- s - string
- a - array
- o - object
- v - void

- . - variable argument list

Each character responds to each type and it allows `ferite` to make sure that the function gets passed the correct parameters. To make life slighty clearer, here are a few examples with the `ferite` function and what would be the equivelent description for a native function:

```
function ex1( string name, number age ){ } would be "sn"

function ex2( string format, ...  )  { } would be "s."

function ex3( object res, string query, array args ) { } would be
"osa"
```

To register the function structure you have, you either use `ferite_register_ns_function` or`ferite_register_class_function`. You must be aware that you can only register each created function once! Otherwise `ferite` will certainly die when it tries to clean everything up at the end of execution.

So, lets assume that our above print function takes a string and a number and prints out the string the number of times it is told. The example below will show how to declare, create and register a FeriteFunction in a namespace. The example will also allow us to touch on another couple of important areas.

```
FE_NATIVE_FUNCTION( printfnc ); /* Declare the prototype */

FE_NATIVE_FUNCTION( printfnc )
{
 FeriteString *print = NULL;
 double countd = 0;
 int i = 0, count = 0;

 /* Get the parameters */
 ferite_get_parameters( params, 2, &print, &countd ); /* #1 */

 /* Loop round printing */
 count = (long)countd;
 for( i = 0; i < count; i++ )
  printf( "%s", print->data );

 FE_RETURN_VOID; /* #2 */
}

void module_init( FeriteScript *script )
{
 /* Create the function */
```

```
  FeriteFunction *f = ferite_create_external_function(
            script,
            "printfnc",
            printfnc,
            "sn" );

  /* Now register it in the main namespace */
  ferite_register_ns_function( script, script->mainns, f );
}
```

Point *#1* is the main point to be covered.: `ferite_get_parameters` is a helper function for getting the values of the parameters into C variables you can manipulate. It is very important that you do not delete or free the values you have because they point to the real values. The first is the parameter list you are given and when writing the native function, it is always called `params`. The second argument is the number of values from the parameter list that you want. The rest of the arguments are pointers to the local C function variables you wish to set. In our example above, the address of the `print` and `countd` variables were passed. This is exactly how `builder` gets the values from the parameter list - it is simply hidden from the programmer.

All functions must return something even if it is just a void. Builder hides *#2* from you, but when writing functions from scratch, it is important you remember to return something.

To get the number of parameters that were passed to the function, you can use the `ferite_get_parameter_count`. This takes just one argument (`params`) and returns the number of variables in it.

```
int ferite_get_parameter_count( FeriteVariable **list );
```

To get the container of the function passed into it, you should use the following macros:

```
#define FE_CONTAINER_TO_OBJECT (FeriteObject*)__container__
#define FE_CONTAINER_TO_CLASS (FeriteClass*)__container__
#define FE_CONTAINER_TO_NS (FeriteNamespace*)__container__
```

These are defined in `ferite.h` and can be used like follows:

```
FE_NATIVE_FUNCTION( toString )
{
  FeriteObject *self = FE_CONTAINER_TO_OBJECT;
  FE_RETURN_CSTR( "Example-toString()", FE_FALSE );
}
```

5.2 The Rest

All you have to do to fullfill the requirements of a `ferite` module is write four functions. These functions are the ones that `builder` creates for you from `module-init, module-deinit, module-register, and module-unregister`.

```
void modulename_register()
{
  /* System wide setup. Called when the
  module is loaded from disk. */
}

void modulename_init( FeriteScript *script )
{
  /* Per script setup. This is where you put the
  code to register namespaces, classes, functions and
  variables and setup anything the script needs. */
}

void modulename_deinit( FeriteScript *script )
{
  /* Anything you need to shutdown per script.
  Ferite will clean up all structures you have registered
  so you do not need to clean those up yourself [eg. the
  namespaces you have registered]. */
}

void modulename_unregister()
{
  /* System wide shutdown. This gets called when
  the ferite engine is being deinitialised. */
}
```

If you have these four functions exported from your module, `ferite` should find them without problem. One thing to note: the name of the module **must** be the same as the prefix for each of the functions otherwise `ferite` will not be able to find them. For instance in `foo.lib` the init function must be called `foo_init`.

You may also want to read the next section as a cunning secret is told that can make writing native modules easier.

6 Embedding Ferite

This section is split into three sub-sections. The first deals with getting the engine up and running within your application so that scripts can be executed. The second section deals with the most effcient way of exporting the application's interface into a script so that useful things can then be done. The third is how to cheat with `builder` and applications.

6.1 Getting The Engine Purring

Ferite is designed to be placed in pretty much anywhere. Therefore it is pretty easy to get the engine up and running, scripts compiled and then executed, and to clean everything up afterwards. To explain how to do this, an example is listed below and afterwards each line is discussed. It is a simple program that shows most of the functionality of the `ferite` command line program.

```
#include <stdio.h>
#include <stdlib.h>
#include <ferite.h>

int main( int argc, char **argv )
{
 FeriteScript *script;
 char *errmsg = NULL, *scriptfile = "test.fe";

 if( ferite_init( 0, NULL ) )
 {
  ferite_add_library_search_path( XPLAT_LIBRARY_DIR );
  ferite_set_library_native_path( NATIVE_LIBRARY_DIR );

  script = ferite_script_compile( scriptfile );
  if( ferite_has_compile_error( script ) )
  {
   errmsg = ferite_get_error_log( script );
   fprintf( stderr, "[ferite:  compile]\n%s", errmsg );
  }
  else
  {
   ferite_script_execute( script );
   if( ferite_has_runtime_error( script ) )
   {
    errmsg = ferite_get_error_log( script );
    fprintf( stderr, "[ferite:  execution]\n%s", errmsg );
```

```
    }
   }
   if( errmsg )
    ffree( errmsg );
   ferite_script_delete( script );
   ferite_deinit( );
  }
  exit( 0 );
}
```

And now, the explanation. It should be noted that only the lines that are critical to the operation of `ferite` will be dicussed, anything that is standard C will be left out.

```
#include <stdio.h>
#include <stdlib.h>
#include <ferite.h>
```

The above is all pretty standard issue. You dont need the `stdio.h` or `stdlib.h` headers to be honest. But with any program they are good practice. The one you do need is `ferite.h`. This will pull all the function prototypes and defines into the program so that the magic may begin. This will ensure that all the headers that are required to interface with ferite are visible to the compiler.

```
if( ferite_init( 0, NULL ) )
```

This line initialises the engine. You must do this before you do anything `ferite` related. This is because this call will initialise the `ferite` memory system, the module system, the regex engine and potentially more things. The prototype for this function looks like this:

```
int ferite_init( int argc, char **argv )
```

If you don't call this it is likely that your program will crash and do various other undefined things. You may call this multiple times and it won't cause issues. It takes two arguments. The first is the number of elemetns contained in the argument array. The second is an array of strings. This is how options are passed into the engine. For a full list of the options that the ferite engine accepts, please look at the command line program's help option. The next step is to setup the paths that ferite looks for when seraching for a module.

```
ferite_add_library_search_path( XPLAT_LIBRARY_DIR );
ferite_set_library_native_path( NATIVE_LIBRARY_DIR );
```

Ferite does what it is told; One of the things that makes it very useful is the ability to control what modules are availible to be loaded. You can obtain the system wide defaults using the **ferite-config** shell script. If you do not call these then the `ferite` engine will be unable to load any of the core modules and will only have the API that the application exports. This is useful for both controlling what the scripters can do and

preventing people from loading rogue modules into the system. With the library paths setup, the next task is to ask ferite to compile a script for us.

```
script = ferite_script_compile( scriptfile );
```

This line will compile the script that is in the file in the `scriptfile` variable. It will always return a script object. The return will either contain the error information or will be an executable script. It is also possible to compile a string into script. For this you call `ferite_compile_string`, it takes one argument which is the script to compile. There are also two more functions, `ferite_script_compile_with_path` and`ferite_compile_string_with_path`, they both take the same arguments as their respective counterparts, with the exception of an added argument. This is a null terminated array of search paths to add to the module system for the duration of the compilation. For more in depth information about these two functions please refer to the C API. For reference here are their prototypes:

```
FeriteScript *ferite_compile_string( char *str );
FeriteScript *ferite_compile_string_with_path( char *str, char **paths
);
FeriteScript *ferite_script_compile( char *filename );
FeriteScript *ferite_script_compile_with_path( char *filename, char
**paths );
```

Having compiled the script, it is important to check to see if any errors have occured:

```
if( ferite_has_compile_error( script ) )
{
  errmsg = ferite_get_error_log( script );
  fprintf( stderr, "[ferite:  compile]\n%s", errmsg );
}
```

This is how we check that everything is working. Or not. `ferite_has_compile_error` will return `FE_TRUE` if there was a compile error and `FE_FALSE` if not. If there is an error, the script will not be executable but you will be able to get the error logs from the script as shown above. You will need to, when finished, delete the script, and still clean up the engine. You will also need to free the string returned using `ffree`.

```
ferite_script_execute( script );
```

If the program execution has got this far you will be wanting to run the script. You pass the script to `ferite_script_execute` which will execute the script. The return value is the return value from the script's main function. To find out whether a runtime error occured you will need to use the code below. **NOTE:** you can run scripts multiple times but it is not recommended, as the state of the script can not be guaranteed. As with compilation, we need to check for errors after we have run the script:

```
if( ferite_has_runtime_error( script ) )
{
  errmsg = ferite_get_error_log( script );
  fprintf( stderr, "[ferite:  execution]\n%s", errmsg );
}
```

`ferite_has_runtime_error` will return true if there has been a runtime error on the script. To get the messages about the error you will need to use `ferite_get_error_log`. This will return the error log as a C string. You will need to free the string returned with `ffree`.

```
if( errmsg )
  ffree( errmsg );
```

Remember to free things!

```
ferite_script_delete( script );
```

Once you have finished with your script object you must delete it. This is a call to `ferite_script_delete`.

```
ferite_deinit( );
```

It's been a long day, you've been running scripts and it's now time to pack your bags and go home. There is one last thing to be done - tell `ferite` to deinitialise. This is done doing `ferite_deinit`. This will cause all allocated memory via `fmalloc/fcalloc/frealloc` to be deallocated, shutdown the module system and anything else that needs to be done. Once this has been called you can re-initialise the system with`ferite_init` and start all over again.

So there you have it. That's how easy it is to get things up and running. It is suggest that you have a look at the command line program in the `ferite` distribution for more options availible or a more concrete example.

6.2 Fake Native Modules

Fake native modules provide a mechanism to export API from the current program to be included when the script is compiled. This allows your scripts to talk to your program when they are being compiled. The example below assumes the methods `theapp_*` compiled into the application.

```
ferite_module_register_fake_module( "theapp.lib",
        theapp_register,
        theapp_unregister,
        theapp_init,
        theapp_deinit );
ferite_module_add_preload( "theapp.lib" );
```

The `ferite_module_add_preload` is important so that the module gets compiled into the script at compile time and therefore allows for initialisation code that gets executed to access the application. Please note that the **.lib** extension is very important. This is so that `ferite` knows that it is a native module and can handle it correctly (and also find it). You should refer to the last section on writing native modules by hand for the information on how to write the native module. For those of you feeling slightly more lazy, read on, there is a cunning use of `builder`.

The above code should be placed before the calls to `ferite_script_compile` or `ferite_compile_string`.

7 Building Modules

The `ferite` distribution currently ships with two mechanisms for building and deploying modules to be run on a system with ferite installed. The first, `generate-module` is a solution based upon the GNU autotools chain, this is considered to be the old and broken way of doing things due to the huge effort required to make sure that it is compatible with the, on average, 5 different versions that ship with the different operating systems. The second, `farm`, is a build system written in `ferite` to provide a sane way of building libraries, modules and binaries. This method is the prefered method of deployment but doesn't currently work on any platform other than Mac OS X and Linux.

At the moment, it is recommend that you ship modules using both methods such that the end user may choose which path to take.

7.1 Most Compatible Method: generate-module

`ferite` ships with the command line program called generate-module. This program takes the name of a ferite module (ending in `.fec`), and builds a autotools based distribution for you. Lets assume we have a `f.fec` module file, and a couple of utility c files that provide helper functions for the code. To execute generate-module we do the following:

```
generate-module f.fec utility.c utility.h
```

Assuming everything is ok, generate-module will create us a directory `f`, and build a autotools distribution. It will copy all the named files provided on the command line into the distribution. To let you know this is happening, you will get an output like this:

```
Ferite Module Distributor (1.0)
Copyright (c) 2000-2002:

Chris Ross <chris@ferite.org>
Sveinung Haslestad <sveinung@cention.se>
Stephan Engstrom <sem@cention.se>

ferite-config:  /opt/local/bin/ferite-config
builder:  /opt/local/bin/builder
prefix:  /opt/local
Using file 'f.fec'
Module name 'f'
Creating Directory 'f'
Reading directory:  /opt/local/share/ferite/generate-module/skel
Reading file:  .
```

```
Reading file:  ..
Reading file:  .cvsignore
Copying .cvsignore to f/
Reading file:  AUTHORS
Copying AUTHORS to f/
Reading file:  autogen.sh
Copying autogen.sh to f/
Reading file:  ChangeLog
Copying ChangeLog to f/
Reading file:  config.h.in
Copying config.h.in to f/
Reading file:  configure.ac
Copying configure.ac to f/
Reading file:  CVS
Reading file:  Makefile.am
Copying Makefile.am to f/
Reading file:  Makefile.in
Copying Makefile.in to f/
Reading file:  README
Copying README to f/
Reading file:  stamp-h.in
Copying stamp-h.in to f/
Reading file:  udcl.sh
Copying udcl.sh to f/
Copying module source over...
Copying module source over...
Copying module source over...
Running builder...

Finished!, now to build a tarball run:

 cd f && ./autogen.sh && make dist

People will then be able to configure and install the tarball

If you wish to add any special checking to to the configure script
please edit f/config.m4
```

To use the module and generate a configure script, change into the directory and run `./autogen.sh`.

7.2 New Method: farm

At the moment, generating a module from a ferite module code is trivial with farm. To tell farm what to do, it is necessary to generate a `farm.yard` file with the module descriptions. The following is a template detail how to build a module and have it installed correctly. You may copy and paste the template into a file, replace all occurances of `MODULENAME` with the name of your module and you should be good to go.

```
<?xml version="1.0" ?>
<yard name="MODULENAME">

 <module id="MODULENAME">
  <list type="source">
   <file name="source/MODULENAME.fec" />
   <fileset dir="source" match="utility\.[ch]$" />
  </list>
  <property type="C">
   <program-output program="ferite-config" arguments="--cflags"
/>
  </property>
  <property type="LD">\
   <program-output program="ferite-config" arguments="--libs" />
  </property>
  <property type="prefix" value="$(FeriteModuleNativeDir)" />
 </module>

 <phase id="install" depends="build">
  <perform action="install" target="MODULENAME" />
  <copy file="source/MODULENAME.fec" target="$(FeriteModuleSourceDir)"
/>
  <copy file="$(ProductDir)/MODULENAME.xml"
    target="$(FeriteModuleDescriptionDir)" />
  <execute program="feritedoc" arguments="--regenerate" />
 </phase>

</yard>
```

The above farm.yard file describes the module, what files exist for the module and the various properties that are required to build it. To use the farm file, you simply invoke `farm build` to build the contents, and `farm install` to install the built products. Everything generated by farm is always placed in the `FarmYard` directory that can be found in the same location as the farm.yard file.

This has been a very quick run through of the use of these tools. As farm mature much more documentation will be released, however, it is still young (but very usable) and

provides a great mechanism for deploying code to a system that has a ferite runtime on it.